AD-A063 588 SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGE--ETC F/G 9/2
THE MESSAGE HANDLER OF ACS.1.(U)
JAN 79 D SAGALOWICZ N00014-77-C-0308
UNCLASSIFIED TR-16 NL

| OF |
AD A063588

END
DATE FILMED
3-79
DDC

LEVEL II

# THE MESSAGE HANDLER OF ACS.1

Technical Report 16

SRI Project 6289
Contract No. N00014-77-C-0308

January 1979

By: Daniel Sagalowicz, Computer Scientist
Artificial Intelligence Center
Computer Science and Technology Division

Prepared for:

Office of Naval Research
Department of the Navy
Arlington, Virginia 22217

Attention: Marvin Denicoff, Program Director
Contract Monitor
Information Systems Branch





SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MNP
TWX: 910-373-1246

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>Technical Report 16 ✓ | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>The Message Handler of ACS.1 | | 5. TYPE OF REPORT & PERIOD COVERED |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Daniel Sagalowicz | | 8. CONTRACT OR GRANT NUMBER(s)<br>N00014-77-C-0308 ✓ |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>SRI International ✓<br>333 Ravenswood Avenue<br>Menlo Park, CA 94025 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>NR 049-308 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Office of Naval Research | | 12. REPORT DATE<br>January 1979<br>13. NO. OF PAGES<br>60 |
| 14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>Unclassified<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Management, knowledge-base, modularity, planning, replanning coordination, organization, model process model, demons., and communications.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the Message Handler, a component of the experimental system Automated Command Support (ACS.1). ACS.1 is intended as a vehicle to develop techniques for building knowledge-based systems that will provide intelligent support to a manager in the areas of planning operations, administration and monitoring of approved plans, and retrospective analysis of completed operations.

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

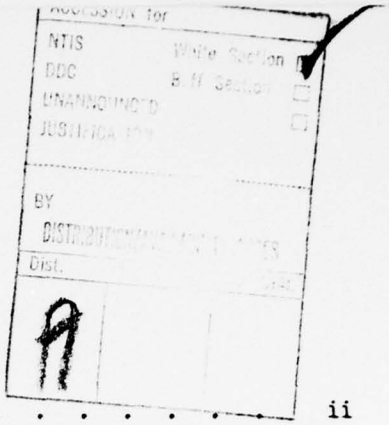
19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

Viewed from the top level, ACS.1 can be regarded as an assembly of modules called "schedulers" and "planners." The planners create detailed plans to meet specified objectives, including the timing of all required tasks and the assignment of all necessary resources. The schedulers coordinate resources, whether people, equipment, supplies, or facilities, so that conflict with other plans or expected events is avoided.

The planners and schedulers work independently. Each has models of its respective tasks and relationships with each other. Global plans are created when these modules communicate with each other via messages. A special component, the Message Handler, routes all the messages, and is at the system's core. Because of its special position, the Message Handler of ACS.1 plays an important role in ACS.1; the user may utilize the message handler to significantly modify the system's behavior. The central idea is to implement a data structure, called the Message Table, which would contain the knowledge of the Message Handler. With this table, this component can modify the messages exchanged by the ACS modules and therefore, totally change the system's behavior.

In this report, we describe the message handler and ways the user can modify the system with it.




CONTENTS

ILLUSTRATIONS

TABLES

## ACKNOWLEDGMENTS

The work described here has been done in collaboration with Marshall Pease. Early contributions to this work were made by Richard Fikes. Jack Goldberg is project supervisor. This research was supported by the Office of Naval Research, Department of the Navy, Arlington, Virginia 22217, under contract N00014-77-C-0308.

# I INTRODUCTION

This report describes the Message Handler, a component of the experimental system Automated Command Support (ACS.1). ACS.1 is intended as a vehicle to develop techniques for building knowledge-based systems that will provide intelligent support to a manager in the areas of planning operations, administration and monitoring of approved plans, and retrospective analysis of completed operations.

Viewed from the top level, ACS.1 is a system of autonomous modules, some of which are called "planners" and some "schedulers," communicating with each other via messages. The planners have the responsibility of planning specified types of activities. The schedulers are responsible for coordinating the use of particular types of resources, which may be people, equipment, supplies, or facilities. The messages sent by the various modules are routed by a special component called the Message Handler. Besides just sending messages to their appropriate destination, the Message Handler may also modify them, in a way specified by the user. The purpose of giving the Message Handler such an active role is to provide the user with a powerful tool to modify drastically the system behavior. This report addresses the design of the Message Handler, describing the other components of the system only to the extent necessary to understand the requirements for the Message Handler.

ACS.1 has operated in the simulated environment of a naval air squadron, although its techniques lend themselves to a wide variety of applications. The principal operations being planned and managed are flight missions. This requires coordinating such various resources as pilots, aircraft, maintenance crews, deck crews, launch facilities and crews, and recovery facilities and personnel. The use of these resources is further limited by such things as the need of personnel for

rest, or of equipment for maintenance. Still unexpected other events can limit the availability of certain resources. For example, a pilot may become sick or an aircraft may require unexpected maintenance. Thus the development and maintenance of a plan to meet specified conditions requires taking into account extensive, complex constraints.

ACS.1 can handle an arbitrary number of resources and an arbitrary number of operations using them. In our current system, the models for these resources and operations are contained in virtual memory, and therefore the main system's limitation is the size of the address space. On PDP-10, the maximum size is 250 pages of 512 words of 36 bits each. The system could easily be extended to store those models on secondary storage, thereby providing for an unlimited number of resources and operations.

Each different application requires appropriate specific design, taking into account different levels of detail and different policy constraints. The tailoring of the system is obtained by specifying the appropriate models.

Some constraints are imposed by limitations in the various types of resources. Other constraints occur when other commitments for these resources are made. The schedulers coordinate the use of these resources. Technical Report 14, "The Schedulers of ACS.1" [2] describes the detailed design of the schedulers of ACS.1.

Other constraints, imposed by the complexities of the operation being planned or administered, are the responsibility of the planners, whose detailed design has been described in Technical Report 15, "The Planners of ACS.1." [3].

Note that the constraints interlock. Although a planner seeks to accommodate those constraints caused by the complexities of the operation, it must do so without conflicting with other commitments made for the required resources. Further, the planner may not recognize that a potential conflict exists until it has completed much of the planning process. Resolution of the resultant conflict by a scheduler may

invalidate part of the previously developed plan. The planner must then revise accordingly. Thus developing a complete plan involves not only the ability to meet its internal constraints, but also the ability to respond to unexpected external conditions that may force replanning of all or part of the plan.

In ACS.1 each module--planner or scheduler--enforces its own set of constraints independently. The overall planning operation is obtained by having the modules communicate with each other via messages. The message handler's chief function is to route the messages from one module to another. In this implementation, ACS.1 imitates quite closely the way a human organization operates, the Message Handler taking the role of the telephone switching system. However, the message handler performs an additional function--namely to facilitate the capabilities of the decision maker to maintain, modify and expand the system's behavior.

This additional function permits the user to have full control of the system operations and the system itself the possibility of modifying its own behavior. This last possibility is not used in the current system; however, we plan to use it in our future research so that "higher level" components can direct the system to adapt itself automatically to new circumstances--environmental changes such as weather, or others such as the existence of other cooperating systems. Those components will be able to use the Message Handler to automatically modify the system's behavior.

The system design and one specific implementation have been described in Technical Report 13, "ACS.1: An Experimental Management Tool" [1], and also in Mr. Pease's paper which has been published in the IEEE transactions on Systems, Man, and Cybernetics [4]. In those two publications, the relationship of this work to other research in artificial intelligence and in optimal scheduling is discussed. That material will not be repeated in this report, although a brief overview of the system design is given in Section II to place the requirements for the message handler in context.

In Section III, a general discussion of the requirements for the Message Handler is given in order to define more precisely what is needed. In Section IV, the detailed design of the message handler is presented, followed in Sections V to IX by a detailed description of each of the major subroutines.

## II SYSTEM DESIGN

The system design of ACS.1 is shown in Figure 1 in block diagram. The main elements of the system are modules called planners and schedulers, each of which is responsible for a well-defined part of the system's operation--a planner for planning a specific type of activity and a scheduler for coordinating the planned use of a specific type of resource. Interactions among the modules are handled entirely through messages passed through a unit called the message handler. All communications to or from a user, or to or from the data system, also pass through the message handler. The message handler helps maintain the autonomy of the separate modules and provides a central switch for user control of the system's operations.

Between the user's terminal and the message handler is the user interface which provides a pseudo-natural language capability. This interface uses a language facility called LIFER developed by the Artificial Intelligence Center of SRI International. The user interface accepts requests or commands in natural language format.*

At this time, the data system is in rudimentary form. It is intended to be more than a simple repository of data and will monitor the execution of approved plans, checking that required tasks are started and completed according to plan, and that needed resources are available. When a conflict occurs, the data system will initiate the appropriate system action, such as replanning or advising the user of the situation.

Each planner and scheduler uses a body of knowledge unique to it, a body that defines how it is to exercise its responsibilities. The

---

* Information about LIFER may be found in "LIFER: A Natural Language Interface Facility," by Gary Hendrix, Technical Note 135 of the SRI Artificial Intelligence Center.

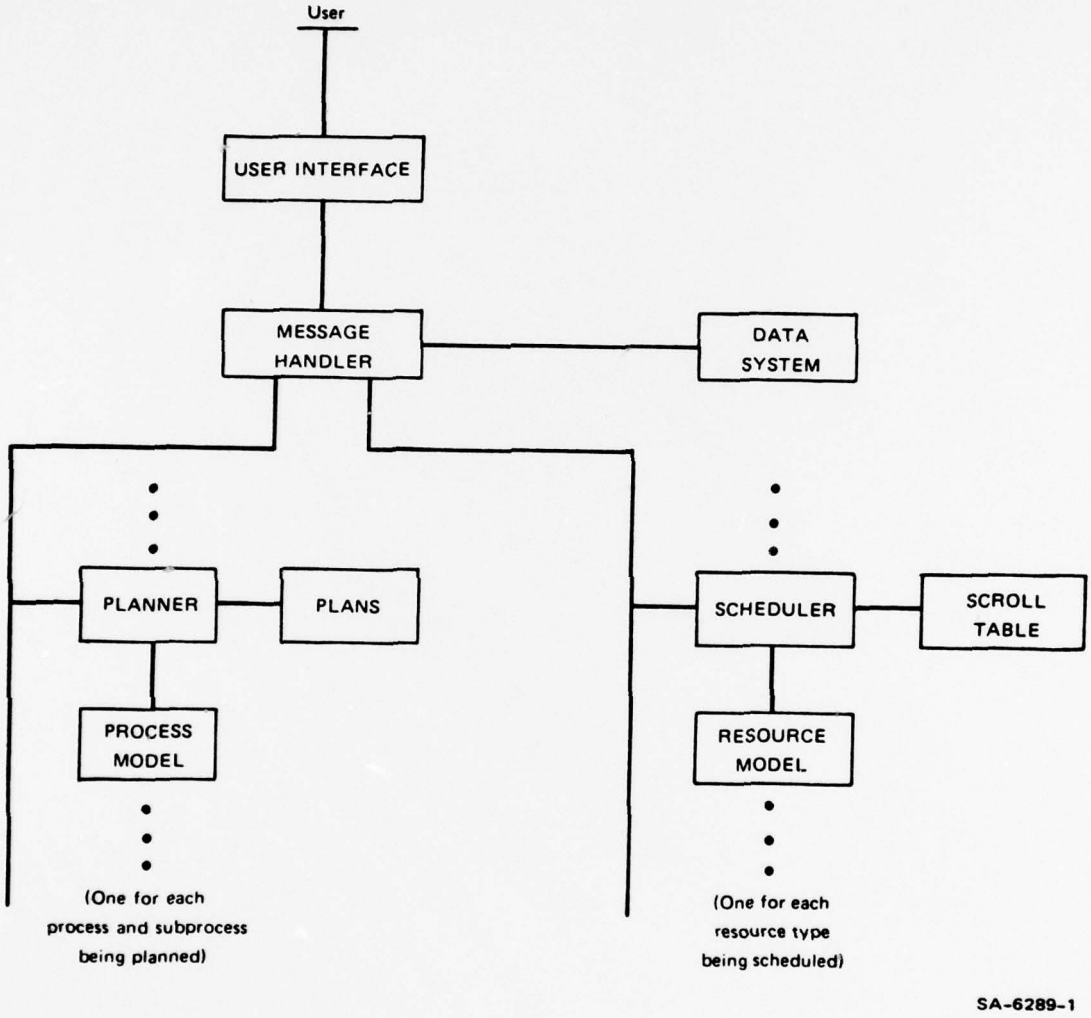



FIGURE 1 BLOCK DIAGRAM OF ACS.I

computer representation of such a body of knowledge incorporated by a planner or a scheduler will be referred to as a "Model" in the rest of this report. The model used by a planner describes the process for which it is responsible to a specified level of detail. It identifies the tasks that compose the process at the given level of detail, and the kinds of resources that must be assigned. It describes the constraints among the tasks--for example certain tasks must be completed before others can be started. It describes the relationship between the assignments of resources and the tasks, such as that an assignment must span the times during which certain tasks are being done. It also includes how an assignment can be obtained, or a plan for a task requested. That is, the model knows how to make requests of the schedulers and other planners, and it knows the identities of other models it must consult to generate a complete plan.

A subtask recognized by a planner as a component in the planner's task may itself require planning. If so, some other planner has the responsibility for planning the subtask, perhaps decomposing it further into subtasks and obtaining additional assignments. The system of planners, in responding to some particular requirement, may be structured into a hierarchy of modules operating at various levels of detail. Different hierarchies may be required in response to different requirements.

Each scheduler coordinates a type of resource, whether people, equipment, supplies, or facilities. When asked for one of the resources of its type to be assigned for some future interval of time, the scheduler first determines a resource's availability. If more than one is available, and if the scheduler has been given the authority, it selects one and returns the name. It also enters the commitment into its own data structure in which it maintains its updated knowledge of the resource commitments. If the scheduler has not been given the authority, it sends the relevant information to the manager for his decision. If no resource is available, the scheduler will either return the closest available assignment or refuse the request, depending on the responsibility given to it.

For example, consider the application that has been studied--the command of a naval air squadron. The principal activity that needs planning is flying a mission. The commander may specify the mission's exact departure from and return to the ship. The requirement is accepted by the planner in charge of planning missions. The process model for that planner--i.e. the computer representation of the knowledge incorporated in that planner--may decompose the activity as shown in Figure 2. The tasks the Mission Planner recognizes are the aircraft's preflight preparation, the pilot's briefing, the flight itself, the aircraft's postflight servicing, and the pilot's debriefing. The process model also includes the fact that a pilot and an aircraft must be assigned. The sequential constraints among the tasks, and the concurrency constraints between the assignments and the tasks that are indicated in Figure 2 are also contained in the process model.

The process model suggested by Figure 2 implies a particular level of detail. The task of preflight preparation of the aircraft, for example, may be further decomposed by another planner into the transfer of the aircraft to the flight deck, its preflight service, arming, fueling, and its launching. Additional resources, such as maintenance personnel, may be required during some of these subtasks, and that information would also be included in the process models.

A generated plan will be returned to the commander for his approval or modification, only after all tasks and subtasks have been planned and all resources needed during any task or subtask have been assigned.

Several features of the system design are considered vital for an easy user interface--both for the system's use and for the model's definition. These features, which have dominated the design of the experimental system, have been significant components of the research. They can be summarized as follows:

* The division of responsibilities among the planners and schedulers should correspond to the division of responsibility in the comparable human organization. This division facilitates the user's understanding of the system's operations, and permits an orderly growth of the system. It also facilitates the transfer of responsibility between the system and the human organization when needed.

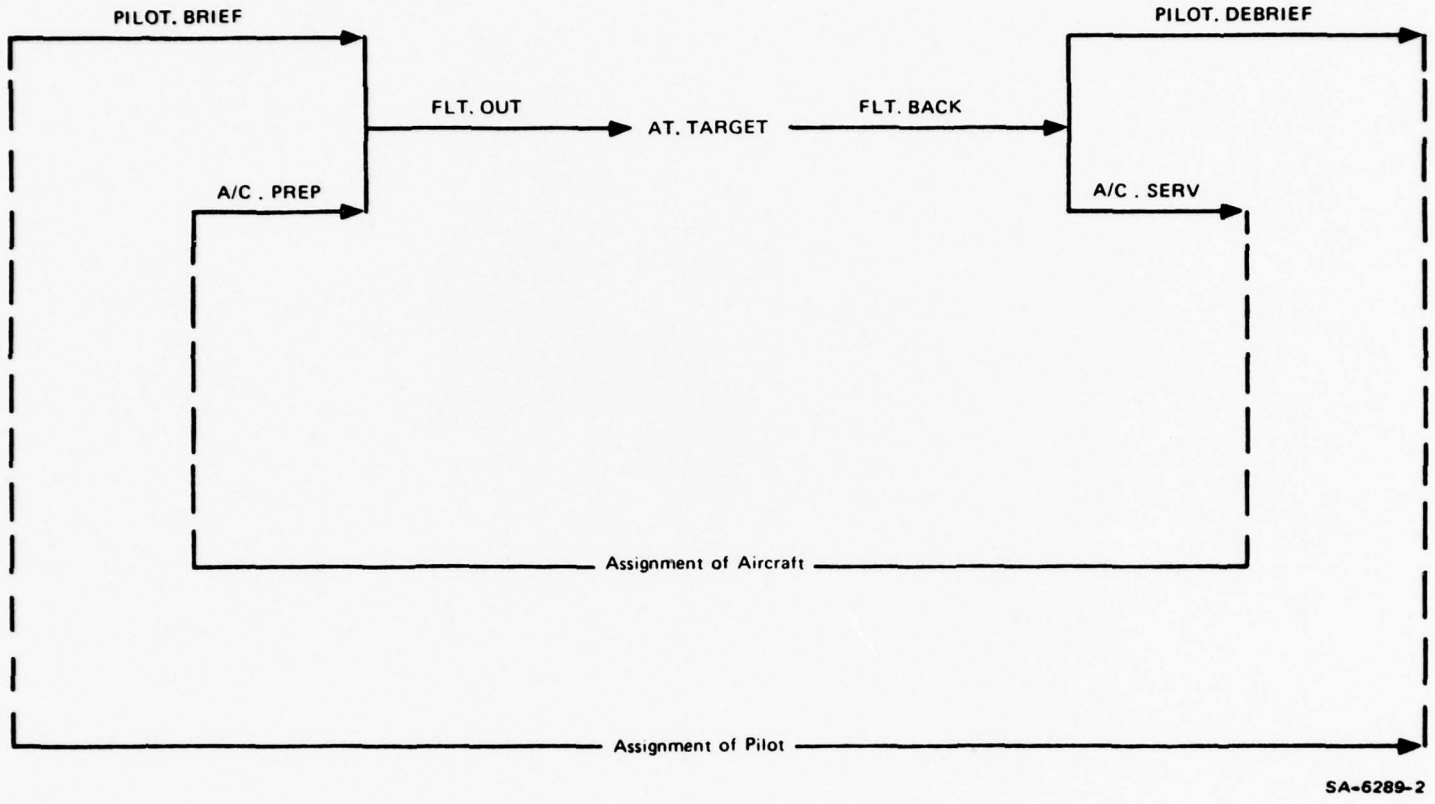



FIGURE 2 PROCESS MODEL FOR FLYING A MISSION

* The models contained by the planners and schedulers should be explicit and accessible for modification without major revision of the system. This is necessary to permit adapting the system to changing needs. It also permits introducing new planners or schedulers through specification of the applicable models. This feature permits the rapid expansion of the system or its transfer to new environments.

* The operation of each planner or scheduler should be sufficiently simple to make it readily understandable by the human user. The complexity of system operation should be the result of interactions among the modules, rather than contained within any module. Again, this facilitates growth and adaptation, and permits rapid modification to meet exceptional conditions.

ACS.1 has been implemented on a PDP-10 using INTERLISP under TOPS-20. It uses a simulated clock in an interactive mode in order to simulate an operational environment.

Further details of the system design, and of the techniques that have been used to implement it, have been given in Technical Report 13 [1] cited earlier. In the next section, we consider the requirements for the message handler in greater detail.

# III MESSAGE HANDLER REQUIREMENTS

The principal system function served by the Message Handler is to route messages from one component of the system to another, i.e. from a planner to a planner or a scheduler, or from a scheduler to a planner. For this function to be executed properly, each message explicitly contains information about where it is coming from, where it is going to, what its main purpose is, and when it is expected to be received by its destination. However, since the Message Handler occupies such a central position in the overall system architecture, it becomes possible to change the way the system behaves by influencing the way the Message Handler acts.

The easiest way to modify the system behavior is to change the contents of the various messages allowing the following possibilities:

* The destination of some messages may be changed. By doing so, one can easily

  dynamically replace a module by a new one. All messages sent to the old module will be sent to the new one,

  replace a module by the user. This allows the user to test new ideas, without having to program them or intervene in exceptional circumstances. All messages sent to the module are now sent to the user, who answers them as if he were the new module,

  implement a complete or partial trace of some or all of the modules. Messages to some or all of the modules are sent to a "tracer-module" which logs them on some logging file(s), then forwards them to their previous destination.

* The text of some messages may be changed. This may be useful

  to temporarily modify a module's behavior. For example, if a new, inexperienced crew replaces an old one, one could increase all expected timings for that particular crew alone and avoid introducing a new process model corresponding to that crew alone,

to ease the modifications to the system. Let us assume that a module, say module A, is to be replaced by a more sophisticated one, module A', which takes into account new information not available to module A. Then, in classical system design, one would first have to replace all modules which call on A to provide that additional information, or at least provide the corresponding default values. With our design, we can either replace all the other modules one by one, and suppress the additional information from messages going to A, until all the modules have been implemented--at which point we replace A by A'. Or, we could start by replacing A by A', and add to all messages going to A' the default values for the missing pieces of information. Clearly, that technique will allow the progressive, dynamic modification of the whole system as required.

The above list is not exhaustive, but gives an idea of the possibilities of this architecture.

This message modification capability has been implemented in the Message Handler. The user fully controls which messages are modified and how they are modified by indicating them explicitly in the "Message Table," as will be explained in section IV.C .

The next section describes the design of the Message Handler and lists the functions used to obtain the desired behavior.

## IV DESIGN AND IMPLEMENTATION OF THE MESSAGE HANDLER

The Message Handler, as described here, is intended to operate in conjunction with the planners and schedulers as described in references [2,3]. Hence, all the functions detailed in thoses references are assumed to be available. In particular, the functions used there for manipulating A-lists--i.e., lists of property-value pairs--are freely used here, without description.

A message is an A-list of the form

```
[(MES.ID . <integer>)
 (FROM . <module-name>)
 (TO . <module-name>)
 (MES.FUNCTION . <function-name>)
 (TIME . <integer>)...]
```

Table 1 shows a sample message taken during an ACS.1 session, and the content of a typical message addressed to the pilot scheduler.

Table 1

Typical Assignment Request Message

| | |
|---|---|
| MES.ID | 5 |
| TO | PILOT.SCHEDULER |
| FROM | MISSION.MODULE |
| MES.FUNCTION | SCHEDULER.ASSIGN.MES |
| TIME | 0 |
| RETURN.MES.TO | RESOURCE.ASSIGNED |
| ARGS | (NAME START END) |
| EET | (850 (PRIORITY . 2)) |
| LST | (380 (PRIORITY . 1)) |
| ... | |

In this example the message contains the following information:

MES.ID is the message number--in this case, this was the fifth message handled by the message handler.

TO is the destination of the message--here the pilot scheduler.

FROM is the source of the message--here the mission module.

MES.FUNCTION is the function that should handle this message for the destination module--here it is a request for a resource assignment. Note that the same module may have several functions. For example, a scheduler is in charge of a specific resource type, but may still perform several functions associated with the assignment of resources, such as resource assignments, resource status reporting, resource status saving on file, and so on.

MES.FUNCTION indicates what function is expected.

TIME is the simulated time when the message should be delivered--here it is 0, i.e., immediately.

RETURN.MES.TO is the function to notify in the sending module. This will become the value of the property MES.FUNCTION in the reply. In this case, the function to be called in the mission module when the reply comes is RESOURCE.ASSIGNED.

ARGS is the list of variables that the mission module expects to see in the reply--here, the name, starting and ending times of the assigned pilot. EET and LST are the earliest ending time and latest starting time demanded for the pilot to be assigned. Other possible indications could be given, such as EST, LET, NAME, START and END which are respectively the earliest starting time, latest ending time, name, suggested actual start time and suggested actual end times. In the present example these were not given and therefore do not appear at all in the message.

Finally, other information may be contained in the message, either for the benefit of the destination module, the benefit of the source module, or just for comment purposes. For reasons of simplicity, we have not included them here.

Table 2 presents the typical response message which corresponds to the previous request for an assignment:

Table 2

Typical Message in Response to an Assignment Request

| | |
|---|---|
| MES.ID | 8 |
| TO | MISSION.MODULE |
| FROM | PILOT.SCHEDULER |
| TIME | 0 |
| MES.FUNCTION | REQUEST.ASSIGNED |
| IN.RESPONSE.TO | ((MES.ID . 5) (TO . PILOT.SCHEDULER) (---)) |
| NAME | ABLE |
| START | 380 |
| END | 850 |

The only significant part which is different from the previous message is IN.RESPONSE.TO. This is a pointer to the data structure representing the original message--in this case message number 5. When it is printed, the whole previous message is printed, but it should be understood that only a pointer is actually part of the reply message.

Until delivery time, the messages are stored in a message queue, which is a priority queue. The delivery time is the priority; and in case of equal delivery times, the value of MES.ID becomes the priority. Before delivery, the messages are modified according to the specifications contained in the message table, as indicated in section III.

# V AUXILIARY FUNCTIONS

The next sections describe the most significant functions which are part of the message handler package. We start with two auxiliary functions, MES.GETP and MES.PUT, which are the equivalent of GETP and PUT for the message A-lists. They take as message input either an A-list or an ID number, in which case the corresponding message A-list is found in the corresponding element of the array MES.HARRAY. MES.GETP gets the value of a property in the message, whereas MES.PUT either puts a new property-value pair or modifies the value of an existing property. The complete definition of these functions is given in Appendix A.

## VI ROUTING FUNCTIONS

In this section, we present the functions which represent the routing role of the message handler. They are CREATE.MES, QUEUE.MES, DEQUEUE.MES and MES.PROCESSOR.

The first function, CREATE.MES, is used to create the message A-list. It takes as input a list of the form

(PROP1 VAL1 ... PROPn VALn),

and creates the corresponding message A-list

[(MES.ID . <integer>)(PROP1 . VAL1)...(PROPn . VALn)],

generating the value of MES.ID automatically. This function also creates an entry in the array MES.HARRAY so that from then on, the message may be retrieved by its ID alone. All the message handler functions may then take as argument either a pointer to the message or the ID of the message--in which case they retrieve the message pointer from MES.HARRAY.

The function, QUEUE.MES, stores the message in the message queue, MES.QUEUE. The message will be processed later by MES.PROCESSOR.

The function DEQUEUE.MES is the inverse of the last one: it removes the message from the message queue, if it has not yet been processed by MES.PROCESSOR. Otherwise, it just prints an error message.

Finally, the last function, MES.PROCESSOR, is in fact the message handler. It is called by the monitor whenever a module releases control. MES.PROCESSOR checks whether any message is in the queue. If so, it modifies it using the message table, as explained in section IV.C, then sends it to its destination. The message is then deleted from the message queue and MES.HARRAY is updated to indicate that the message has been sent.

Table 3, which presents an extract from an ACS.1 session, shows the routing function of the message handler. In this example, the user asks the system to plan for a mission to start in 3 hours.

Table 3

*Message Switching Example*

```
(1)<MONITOR>ttymes)

PLEASE SPECIFY THE MESSAGE PROPERTIES AND VALUES:

   TO              mission.module
   MES.FN          enter.value
   MODEL.NAME      mission
   PLAN.ID         1
   TYPE            tasks
   NAME            flt
   PROP            start
   VAL             500
   PRIORITY
   PRINT.SUP.FLG
THANK YOU.

(2) QUEUE FN.MES MESSAGE 2 TO MISSION.MODULE
(3) GOING BACK TO MONITOR.
(4) SEND FN.MES MESSAGE 2 FROM TTY: TO MISSION.MODULE
(5)    QUEUE PLAN.TASK.MES MESSAGE 3 TO A/C-PREP.PLANNER
(6) SEND PLAN.TASK.MES MESSAGE 3 FROM MISSION.MODULE TO
                                        A/C-PREP.PLANNER
```

In the table, the characters typed by the user are in small letters. The rest is typed to the user by the system--either for information purposes or for requesting needed data from the user.

First the user specifies that he wants to send a message to the mission module (from line 1 to line 2). Then, the user is notified that the message has been queued--using QUEUE.MES (line 2). The TTY module then releases control, and the monitor calls on MES.PROCESSOR, i.e. the message handler (line 3). The message handler sends the message to the mission module--i.e. the control is transferred to the mission module

(line 4). That module queues a message which is due to go to the aircraft preparation planner (line 5), and the scenario will repeat itself, i.e., the message handler sends the message to the aircraft preparation module which then takes control and queues another message to another module, and so on.

The function definitions of all the routing functions are given in Appendix B.

## VII MESSAGE MODIFICATION FUNCTIONS

We now present the functions used to modify the messages. The first one, MES.MODIFY, is called by MES.PROCESSOR. It checks whether entries in MES.TABLE apply to the message.* If so, the message is modified accordingly. Actually, the checking is done by MATCH.MES.WITH.TABLEENTRY, and the message modification is done by MES.APPLY.ENTRY .

MATCH.MES.WITH.TABLEENTRY tries to decide whether the message matches with an entry in MES.TABLE. The entry has the following format:

(<condition> <change specification>)

The role of MATCH.MES.WITH.TABLEENTRY is to check that the message matches the <condition> part of the entry, which has the following format:

```
<condition> := (<cond.prop> <cond.val>)
               or (AND <condition> ... <condition>)
               or (OR  <condition> ... <condition>)
               or (NOT <condition>)
               or (NULL <condition>)
```

A match between a message and a condition of the type (<cond.prop><cond.val>) occurs if one of the following is true:

<cond.prop> is the atom ::

or

the message has a property <cond.prop> and <cond.val> is ::

or

the message has a property-value slot equal to (<cond.prop>. <cond.val>)

or

the message has a property <cond.prop> and the value slot is a list whose CAR is ::. In this case, the CDR of the list is EVALED.

---

* The entries in MES.TABLE are prepared by the user. Although we have not done it, functions could be written to help the user set up MES.TABLE.

Then the message may be modified, or side effects may occur.

The extension to the conditions of the type AND, OR, NOT, or NULL is obvious.

The next function, MES.APPLY.ENTRY, modifies the message according to the <change specification> part of the entry. More precisely, the <change specification> is a list of the form:

(... (<prop> <val>) ...)

If <prop> is NIL, <val> is evaluated--using TABLE.ENTRY.GET-- and the message is not modified directly: presumably, we are only interested in the side effects of evaluating <val>, which may modify the message indirectly. Otherwise, a new slot is added to the message, with property name <prop>, and value the value returned by TABLE.ENTRY.GET .

Finally, TABLE.ENTRY.GET evaluates the entry element, i.e., the <val> part of the <change specification>. <Val> may be one of the following:

an atom, in which case it is simply returned

or

a list whose first element is not ":", in which case the list is returned

or

a list starting with ":", in which case each element of the list is evaluated, and the last value is returned.

As an example of message modifications, consider table 4 which represents an actual session with a user.

Table 4

Message Modification Example

```
(1)<MONITOR>ttymes)
PLEASE SPECIFY THE MESSAGE PROPERTIES AND VALUES:

                TO              m.m
                MES.FN          enter.value
                MODEL.NAME      mission
                PLAN.ID         1
                TYPE            tasks
                NAME            flt
                PROP            start
                VAL             500
                PRIORITY
                PRINT.SUP.FLG
                *

THANK YOU.
(2) QUEUE FN.MES MESSAGE 4 TO M.M
Going back to MONITOR.
(3) SEND FN.MES MESSAGE 4 FROM TTY: TO MISSION.MODULE
(4)    QUEUE SCHEDULER.ASSIGN.MES MESSAGE 5 TO PILOT.SCHEDULER
(5) SEND SCHEDULER.ASSIGN.MES MESSAGE 5 FROM MISSION.MODULE TO TTY:
(6)SCHEDULER.ASSIGN.MES  MESSAGE 5 RECEIVED FROM  MISSION.MODULE
```

Here, the same conventions are used as in the previous table: the user's input is presented in small letters. The user prepares a message to be sent to "m.m" (between lines 1 and 2). The message to m.m is queued (line 2). The message handler sends the same message to the mission module, i.e., the message has been modified by replacing m.m with mission.module (line 3). The mission module queues a message to the pilot scheduler (line 4). The message handler sends that message to TTY:, i.e., to the user (line 5). Here again, the message was modified, replacing PILOT.SCHEDULER by TTY: . Finally, the TTY: module, which is the user interface, notifies the user that a message has been received (line 6).

In Table 5, we present the message table which gave rise to those two modifications:

Table 5

Message Table Example

```
(1)  [(((TO M.M))
(2)    ((TO MISSION.MODULE)))
(3)   ([(NOT ((TO TTY:)))
(4)     (NOT ((TO MISSION.MODULE]
(5)    ([ORIGINAL.TO (:(MES.GETP MES (QUOTE TO]
(6)     (TO TTY:]
```

This message table has two entries. The first entry applies to all messages addressed to M.M (line 1), and modifies them by replacing the TO part of the message by MISSION.MODULE (line 2). The second entry applies to all messages which are not addressed to either the user (line 3) or the mission module (line 4). It modifies those messages by replacing the TO part by TTY: (line 6), and by adding an ORIGINAL.TO part which will contain the original value of the TO part, i.e., the original destination of the message (line 6).

These two entries correspond to a typical case where a user, testing a module alone (here: MISSION.MODULE), wishes to receive all messages coming from this module (second entry), and also wishes not to write out the complete module name everytime he sends a message to it (first entry). Since this case is so typical in system development, a function could be written to create two such entries automatically for any module, and any alias the user would wish to use.

As another example, let us consider the aircraft launch process, and let us assume that it takes one minute to launch an aircraft in good weather, and three minutes in bad weather. The problem is how to implement both conditions easily in ACS.1.

There are basically three ways of achieving this result in ACS.1 First, the user can change dynamically the process model which

corresponds to the flight task. In good weather, that model would indicate that the launch takes one minute; in bad weather the user would modify it to make the launch three minutes long. The only disadvantage of that solution is that the user has to change the models constantly instead of having them prepared in advance.

The second solution is to have both models prepared in advance, and have in essence two flight modules--one which uses the original model, the other the modified model. Then, the user would implement the change of weather by making an entry in the message table which would indicate to what module the flight task belongs--given the current conditions.

The third solution is in the same spirit as the second, but simpler in realization. The basic idea is that whenever a message is sent to the launch facility scheduler by the flight module, the time to be reserved will be increased by two minutes. This solution can again be easily implemented by making an entry in the message table.

We give the definition of all the functions involved in the message modification role of the message handler in Appendix C.

This completes the presentation of the message handler per se. However, we believe that its usefulness can mainly be justified by its advantages to the user. To emphasize this point, section V presents the functions we have developed so far as part of the user interface.

## VIII USER INTERFACE FUNCTIONS

The user's primary need in the user interface is to be able to specify a message to be sent. This need is met by the function TTYMES. It asks the user for the destination of the message and the function requested. Then, it automatically asks for the values of the arguments that the function needs. Finally, it asks the user whether he wants to add any more attribute/value pairs to the message. It then creates the message and sends it via the message handler. When the message arrives at its destination, it is handed to the function FN.MES, which in a sense is part of the user interface although used inside the other modules. FN.MES knows how to call the appropriate function from the content of the messages created by TTYMES. Essentially, it finds in the property ARGS of the message the list of arguments to be part of the function call, and under each argument name finds the appropriate value.

Table 6 presents an example of the use of TTYMES.

Table 6

User Entry of a Message

```
<MONITOR> ttymes)

PLEASE SPECIFY THE MESSAGE PROPERTIES AND VALUES:

          TO              mission.module
          MES.FN          enter.value
          MODEL.NAME      mission
          PLAN.ID         1
          TYPE            tasks
          NAME            flt
          PROP            start
          VAL             500
          PRIORITY
          PRINT.SUP.FLG
comment (this is a message asking that a mission flight
start at 500]
THANK YOU.
```

The first two questions (TO and MES.FN) are always asked by TTYMES. ENTER.VALUE is a function which has eight arguments, namely MODEL.NAME, ID, TYPE, NAME, PROP, VAL, PRIORITY and PRINT.SUP.FLG . Those arguments are requested by TTYMES once it is told which function is being invoked. Finally, TTYMES checks to see if the user wants any more information in the message, and in the example, the user added a COMMENT. In this example, TTYMES created the following message:

```
TO              MISSION.MODULE
FROM            TTY:
MES.FUNCTION    FN.MES
MES.FN          ENTER.VALUE
MODEL.NAME      MISSION
ID              1
TYPE            TASKS
NAME            FLT
PROP            START
VAL             500
PRIORITY        NIL
PRINT.SUP.FLG   NIL
COMMENT         (THIS IS A MESSAGE ASKING THAT A MISSION FLIGHT
                 STARTS AT 500)
ARGS            (MODEL.NAME ID TYPE NAME PROP VAL PRIORITY
                 PRINT.SUP.FLG COMMENT)
```

After receiving this message, the mission module will call FN.MES, since this is the value of the MES.FUNCTION property in the message. It gives FN.MES the message as an argument. In turn, FN.MES will call ENTER.VALUE (the value of the MES.FN property in the message) and give it as argument values the values of the arguments listed in ARGS. Note that the value of the COMMENT property will be given to ENTER.VALUE which will not see it since it only takes eight arguments. In this case, we can consider that COMMENT--and other extra property/value pairs--are ignored.

The next function, PRINT.TTYMES, is called whenever a message for the user is received. It briefly describes the message, and then asks the user what he wishes to do. Depending on the user's answer, one of several functions which will take the appropriate actions may be called. We first show in Table 7 an example of an interaction with PRINT.TTYMES.

Table 7

User Interaction with PRINT.TTYMES

```
(1) SCHEDULER.ASSIGN.MES MESSAGE 5 RECEIVED FROM MISSION.MODULE

(2) WHICH ACTION DO YOU WISH TO TAKE? <?>
ONE OF:
REPLY TO IT
FORWARD IT
PRINT IT
SUSPEND PLAN AND SAVE MESSAGE
DEFER ANSWER (MESSAGE SAVED AND PLAN CONTINUED)
BYPASS QUESTION (PLAN CONTINUED BUT MESSAGE LOST)
<CR> MAY ALSO BE USED INSTEAD OF "R" TO REPLY TO THE MESSAGE.

(3) WHICH ACTION DO YOU WISH TO TAKE? pRINT IT

IT CONTAINS THE FOLLOWING INFORMATION:

MES.ID          5
FROM            MISSION.MODULE
TO              TTY:
TIME            0
MES.FUNCTION    SCHEDULER.ASSIGN.MES
RETURN.MES.TO   REQUEST.ASSIGNED
ASK.TO.TTY      REQUEST.ASSIGN.FROM.TTY
ARGS            (NAME START END)
EET             (850 (PRIORITY . 2))
LST             (380 (PRIORITY . 1))
...
(4) WHICH ACTION DO YOU WISH TO TAKE? sUSPEND

ID FOR SAVED MESSAGE: 1
```

The user is first notified that a message was received (line 1). Then, he asks what his options are by typing a question mark--which is not echoed back (line 2). Knowing his options, he then asks to see the complete message (line 3). Finally, he asks that the planning operation be suspended, and the message saved (line 4). Then, the system asks for an id number for the message--in this case, the user types 1.

In the above example, the user asked that the planning operations be suspended and the message saved. This saving was done by SAVE.MES, which stored the message in an array named MES.SAVED.HARRAY.

Instead of suspending the planning operations, the user could have replied to the message. Also, after having suspended the planning, the user may resume it by answering the saved message. The example in Table 8 presents such an interaction.

Table 8

Example of a User Reply

```
(1)<MONITOR> reply.mes(1)

(2) NEED AN ASSIGNMENT OF A PILOT FOR MISSION NUMBER 1.
CURRENT REQUIREMENTS ARE AS FOLLOWS:
     LATEST START TIME DESIRED:  380.
     EARLIEST END TIME DESIRED:  850.

(3)       ENTER NAME OF PILOT ASSIGNED,
          OR <CR> TO DEFER ASSIGNMENT:  able

(4) ARE THE LATEST END AND EARLIEST START TIMES OK? yES

(5) DO YOU WANT TO ADD ANYTHING ELSE TO THE MESSAGE? NO

(6) QUEUE REQUEST.ASSIGNED MESSAGE 45 TO MISSION.MODULE
GOING BACK TO MONITOR.
```

The user says that he wants to reply to the message saved under ID 1 (line 1). Then, a brief summary of the message is presented to the user (between line 2 and line 3). This is done using the function REQUEST.ASSIGN.FROM.TTY which was indicated in the message in Table 7 as the value of the ASK.TO.TTY part. Then, REQUEST.ASSIGN.FROM.TTY asks the user for the name of a pilot (line 3) and the acceptability of the indicated times (line 4). Finally, the user is asked for other property/value pairs to be added to the message (line 5); he does not add any. The message is then built and sent to the mission module (line 6).

The definition of REPLY.MES will be given in Appendix D, followed by REQUEST.ASSIGN.FROM.TTY as an example of one such "specific function" which may be used to simplify the user interface. Without this function, the user could have built the message directly in a more tedious fashion.

Finally, the user may want to forward the message to either the original destination or a new one. This can be initialized either when the message is first received by the user interface, or later, after the message had been saved. Table 9 presents such an example taken from an ACS.1 session.

Table 9

Example of Message Forwarding Interaction

```
(1)    QUEUE PLAN.TASK MESSAGE 3 TO A/C-PREP.PLANNER
(2) SEND PLAN.TASK.MES MESSAGE 3 FROM MISSION.MODULE TO TTY:
(3) PLAN.TASK.MES MESSAGE 3 RECEIVED FROM MISSION.MODULE

(4) WHICH ACTION DO YOU WISH TO TAKE? FORWARD IT
(5)     TO: a/c-prep.planner
(6) DO YOU WANT THE ORIGINATOR OF THE MESSAGE TO BE TTY:? yES

(7) QUEUE PLAN.TASK MESSAGE 3 TO A/C-PREP.PLANNER
(8) SEND PLAN.TASK MESSAGE 3 FROM TTY: TO A/C-PREP.PLANNER
```

The mission module sends a message to the aircraft preparation planner (line 1). The message is rerouted by the message handler to the user (lines 2 and 3). PRINT.TTYMES informs the user that a message was received (line 3) and asks for further instructions (line 4). The user asks that the message be forwarded (line 4) to the aircraft preparation planner (line 5), and that the user be indicated as the origin (line 6) so that the message will not be rerouted to the user again, but will go instead to the planner itself (line 7 and 8). The function which asks the questions on lines 5 and 6 is FORWARD.MES which also will build the new message.

Appendix D gives the definitions of all the user interface functions.

# IX CONCLUSIONS

The message handler of ACS.1 has added new possibilities for helping the user maintain, modify, and extend the planning and scheduling system. Its main role was originally to imitate the functions of a switching network such as the telephone system, mostly to provide a true asynchronism and independence between modules. However, it has been extended to allow the dynamic modification of messages. As has been explained in the previous sections, this allows for a dynamic, gradual modification of the system behavior. It is believed that this component will facilitate the development and maintenance of a complex planning/scheduling system, at the same time giving the user confidence in using the system, knowing that he can modify its behavior.

For the future, the message handler could become the main tool for a system to modify itself. "Higher level" modules could apply their knowledge of a situation and modify automatically the message table. Also, in a distributed, cooperative system environment the message handler could use its own model of the cooperative modules to "understand" and "translate" the messages to and from the local system.

The idea of a message handler is not new; however the expanded role it plays with its message modification possibilities suggests such a component will to help management perform their functions more efficiently.

## REFERENCES


[1] Pease M., "ACS.1: An Experimental Management Tool," Technical Report 13, Computer Science Laboratory, SRI International, 1977.

[2] Pease M., "The Schedulers of ACS.1," Technical Report 14, Computer Science Laboratory, SRI International, September 1977.

[3] Pease M., "The Planners of ACS.1," Technical Report 15, Computer Science Laboratory, SRI International, November 1977.

[4] Pease M., "ACS.1: An Experimental Management Tool," IEEE Transactions on Systems, Man, and Cybernetics, Vol. 8, No. 10, pp. 725-735, October 1978.

# Appendix A

## Auxiliary Functions

## Appendix A

## Auxiliary Functions

```
(MES.GETP
  [LAMBDA (MES PROP)
    (COND
      ((LISTP MES)                          (* It is an A-list)
        (A*.GETP MES PROP))
      ((A.GETP (GETHASH MES MES.HARRAY) (* It is not: get the)
              PROP])                        (* A-list in the array)


(MES.PUT
  [LAMBDA (MES PROP VAL)
    (COND
      ((LISTP MES)
        (A*.PUT MES PROP VAL))
      ((A.PUT (GETHASH MES MES.HARRAY)
              PROP VAL])
```

# Appendix B

## Routing Functions

## Appendix B

## Routing Functions

```
(CREATE.MES
  [LAMBDA X
          (* This is a LAMBDA nospread function.
          Function: creates an a.list, generates an id, enters
          the id into the a.list; input: CREATE.MES
          (prop1;value1;prop2;value2;...); output: returns an
          a.list as a value such as ((mes.id . <integer>)
          (prop1 . value1) (prop2 . value2) ...) .
          If called with no arguments, the value would have
          the form ((mes.id . <integer>)).)
          (* An association between a pointer to a message
          and the id of that message is stored in the hash
          array called MES.HARRAY such that the pointer is
          retrievable by the id.)

     (PROG (MES (M1 1)
                (M2 2)
                (T0 0)
                T1)
          [SETQ MES (LIST (CONS (QUOTE MES.ID)
                                (SETQ T1 (TRACE.GEN.ID.TEMPORARY]
          (PUTHASH T1 MES MES.HARRAY)


       LP  (if X = T0
              then (RETURN MES)
            else (NCONC1 MES (CONS (ARG X M1)
                                   (ARG X M2)))
                 (SETQ M1 (IPLUS M2 1))
                 (SETQ M2 (IPLUS M1 1))
                 (SETQ T0 (IPLUS T0  2))
                 (GO LP])



(QUEUE.MES
  [LAMEDA (MES)
          (* Takes as input either a message A.list or an
          ID of a message, stores that message in the
          message queue and then returns the ID of the message
          just stored into the message queue.)
     (PROG (T0 T1)
```

```
        [COND
          ((NOT (LISTP MES))                          (* First get)
            (SETQ MES (GETHASH MES MES.HARRAY]        (* the A-list)
          (SETQ T1 (MES.GETP MES (QUOTE MES.ID)))

        (* Then check if the time is there.
          If not: assume immediate delivery)

          (COND
            ((NULL (A.GETP MES (QUOTE TIME)))
              (A.PUT MES (QUOTE TIME)
                     S.CLOCK)))
          [SEM.TRACE Queue ((MES.GETP MES (QUOTE MES.FUNCTION)))
                     message
                     (T1)
                     to
                     ((MES.GETP MES (QUOTE TO]

        (* Traces the message if so required)

          (ADD.QUEUE (QUOTE MES.QUEUE)
                     MES)

        (* Finally, adds it to the message queue)

          (RETURN T1])


(DEQUEUE.MES
  [LAMBDA (MES)
          (* Takes either a message a.list or a message id.
          If the message is not deleted then returns the
          deleted id else returns the message  that the id
          is already deleted.)
    (PROG (T1)
          [COND
            ((NOT (LISTP MES))          (* First, get the A-list)
              (SETQ MES (GETHASH MES MES.HARRAY]
          (SETQ T1 (MES.GETP MES (QUOTE MES.ID)))
          (COND
            ((EQUAL (GETHASH (QUOTE MES.QUEUE)
                             SYSTEM.QUEUE)
                    (DEL.NODE (QUOTE MES.QUEUE)
                              MES))
              (COND
                ((EQ (QUOTE YES)
                     S.TRACE)

        (* The message has already been processed or deleted)
```

```
            (PRIN1 "THE MESSAGE WHOSE ID IS    ")
            (PRIN1 T1)                    (* Say so if tracing)
            (PRINT1
          "  HAS ALREADY BEEN DELETED FROM THE MESSAGE QUEUE.")))
          (RETURN T1))
         (T                              (* delete this message)
            (COND
             ((EQ (QUOTE YES)
                  S.TRACE)               (* and say so if tracing)
              (PRIN1 "THE MESSAGE WHOSE ID IS   ")
              (PRIN1 T1)
              (PRINT1
                 "  HAS BEEN DELETED FROM THE MESSAGE QUEUE.")))
           (RETURN T1])



(MES.PROCESSOR
  [LAMBDA (MES)

          (* Input: either A.list form of a message or an id
          form of the message; output: message id which has
          been processed; function: receives a message and
          applies the module specified by the A.list value
          corresponding to the property TO, to the message passed
          to the module. I.e., it sends the message to the module
          whose name is contained in the slot corresponding to
          "TO")

          (* After it has processed MES, it puts it into a trace
          file (mes.trace) and changes the hash array
          (MES.HARRAY) entry of the message processed into NIL.)

          (* It also checks the MES.TABLE to modify the message
          before sending it)

     (PROG (MES.ID MES.TO MESSAGE TO.WHOM TO.LOCK.LIST MES.LIST)
          [COND
            ((NLISTP MES)       (* We HAVE the message id, get
                                   the whole message A-list)
              (SETQ MES (GETHASH MES MES.HARRAY]

          (* Check if the message matches any entry in MES.TABLE.
          MES.MODIFY will do both the checking and the
          modifications if needed)

          (SETQ MES (MES.MODIFY MES))
          (SETQ MES.LIST (CDR (GETHASH (QUOTE MES.QUEUE)
                                       SYSTEM.QUEUE)))
```

```
L3  (SETQ MES.ID (MES.GETP MES (QUOTE MES.ID)))
    (SETQ MES.TO (A.GETP MES (QUOTE TO)))

          (* trace the message processing if required)

    (SEM.TRACE Send ((MES.GETP MES (QUOTE MES.FUNCTION)))
               message
               (MES.ID)
               from
               ((MES.GETP MES (QUOTE FROM)))
               to
               (MES.TO))
    (SETQ S.TRACE.LEVEL 3)
    (SETQ TO.WHOM (MES.GETP MES (QUOTE IN.RESPONSE.TO)))
    (COND
      [(OR (NULL (SETQ TO.LOCK.LIST
                       (GETPROP LOCK.LIST MES.TO)))
           (MEMB TO.WHOM TO.LOCK.LIST))
        [COND
          (TO.LOCK.LIST (PUTPROP LOCK.LIST MES.TO
                                 (REMOVE TO.WHOM TO.LOCK.LIST]
        (DEL.NODE (QUOTE MES.QUEUE)
                  MES)
        (SETQ MESSAGE (RESUME MONITOR (EVAL MES.TO)
                              MES))

          (* This in effect sends the message to destination)

        (COND
          ((NULL MESSAGE)
            (PUTPROP LOCK.LIST MES.TO NIL))
          ((EQ (CAR MESSAGE)
               (QUOTE *))
            (PUTPROP LOCK.LIST MES.TO (2ND MESSAGE)))
          ((AND (EQ (CAR MESSAGE)
                    (QUOTE ADD))
                (EQ (CAR (2ND MESSAGE))
                    (QUOTE *)))
            (for EACH.MES in (2ND (2ND MESSAGE))
               do (ADDPROP LOCK.LIST MES.TO EACH.MES)))


          ((EQ (CAR MESSAGE)
               (QUOTE ADD))
            (ADDPROP LOCK.LIST MES.TO (2ND MESSAGE)))
          [(AND (EQ (CAR MESSAGE)
                    (QUOTE DELETE))
                (EQ (CAR (2ND MESSAGE))
                    (QUOTE *)))
            (for EACH.MES in (2ND (2ND MESSAGE))
               do (PUTPROP LOCK.LIST MES.TO
                           (REMOVE EACH.MES (GETPROP LOCK.LIST
```

```
                                                        MES.TO]
      ((EQ (CAR MESSAGE)
           (QUOTE DELETE))
        (PUTPROP LOCK.LIST MES.TO
                 (REMOVE (2ND MESSAGE)
                         (GETPROP LOCK.LIST MES.TO]
  (T

(* Do not send a message; pass over the locked out
message and consider the next message on the queue.)

     (COND
       ((SETQ MES (CAR MES.LIST)))
       ((PRINT1
    "No more message to be served.  Error in the system.")
        (HELP)))
     (SETQ MES.LIST (CDR MES.LIST))
     (GO L3)))
(SETQ S.TRACE.LEVEL 0)
(RETURN MES.ID])
```

# Appendix C

## Message Modification Functions

## Appendix C

## Message Modification Functions

```
(MES.MODIFY
  [LAMBDA (MES)

          (* Check if any entry in MES.TABLE applies to the
          message: if yes does the modifications and loops
          back; if no, returns the final result)


     (PROG ((NEWMES MES)
            (MES.TABLE.PTR MES.TAELE)
            MES.TABLE.ENTRY)
       LOOP(COND
             (MES.TABLE.PTR              (* more entries in the table:
                                          check the next one)
                 (SETQ MES.TABLE.ENTRY (CAR MES.TABLE.PTR))


                 (COND
                   ((MATCH.MES.WITH.TABLEENTRY NEWMES
                                               (CAR MES.TABLE.ENTRY))
                    (SETQ NEWMES (MES.APPLY.ENTRY
                                  NEWMES
                                  (CADR MES.TABLE.ENTRY)))
                    (SETQ MES.TABLE.PTR MES.TABLE)
                                          (* Now we loop back, and
                                          start all over again)
                    (GO LOOP)))
                 (SETQ MES.TABLE.PTR (CDR MES.TABLE.PTR))
                 (GO LOOP))
             (T

          (* That's it! NEWMES is the new message; it will be sent
          to the module contained in its TO slot by MES.PROCESSOR)

              (RETURN NEWMES])


(MATCH.MES.WITH.TABLEENTRY
  [LAMBDA (MES MATCHLST ORFLG)
```

```
(* Tries to match the message with the list in MATCHLST.
If ORFLG is T, only one element of the matchlist has to
match; the matchlist may start with AND, OR, NOT or NULL
in which case the appropriate recursive call is made)

(PROG (VAL PROP)
      (RETURN
        (COND
          [MATCHLST          (* There is something to match
                                with: let us try it out...)
            (COND
                [(EQ (CAAR MATCHLST)
                     (QUOTE AND))
      (* The match list starts with an AND: we must match
      everything inside the AND list)
                 (COND
                    (ORFLG
      (* we were called with ORFLG=T, then we must either match
      everything in the AND list, or match some element in the
      OR list--of which the AND list is only one!)
                       (OR (MATCH.MES.WITH.TABLEENTRY MES
                                                (CDAR MATCHLST))
                           (MATCH.MES.WITHTABLEENTRY MES
                                                (CDR MATCHLST) T)))
                    (T
      (* otherwise: we were part of another AND list: we must
      match EVERY thing)
                       (AND (MATCH.MES.WITH.TABLEENTRY MES
                                                (CDAR MATCHLST))
                            (MATCH.MES.WITH.TABLEENTRY MES
                                                (CDR MATCHLST]
                [(EQ (CAAR MATCHLST)
                     (QUOTE OR))       (* We go through the same
                                       type of thing for an OR list)


                 (COND
                    (ORFLG
      (* We were part of an OR list: we must either match this
      OR list, or any other element of the upper OR list!)
                       (OR (MATCH.MES.WITH.TABLEENTRY
                                MES (CDAR MATCHLST) T)
                           (MATCH.MES.WITH.TABLEENTRY
                                MES (CDR MATCHLST) T)))
                    (T
      (* We were part of an AND list: we must match at least one
      element of this OR list, and all the other elements of the
      upper AND list)
                       (AND (MATCH.MES.WITH.TABLEENTRY
                                MES (CDAR MATCHLST) T)
                            (MATCH.MES.WITH.TABLEENTRY MES
                                                (CDR MATCHLST]
```

```
        [(OR (EQ (CAAR MATCHLST)
                 (QUOTE NULL))
             (EQ (CAAR MATCHLST)
                 (QUOTE NOT)))
                        (* This is a NULL or NOT list)
         (COND
            (ORFLG
(* We were in an OR list: either we do not match the
element of this NULL list, or we match some other element
of the upper OR list)
               (OR (NULL (MATCH.MES.WITH.TABLEENTRY
                                  MES (CADAR MATCHLST)))
                   (MATCH.MES.WITH.TABLEENTRY
                           MES (CDR MATCHLST) T)))
            (T
(* We were in an AND list: we must not match the element of
the NULL list, and we must match all the other elements of
the upper AND list)
               (AND (NULL (MATCH.MES.WITH.TABLEENTRY
                                 MES (CADAR MATCHLST)))
                    (MATCH.MES.WITH.TABLEENTRY
                           MES (CDR MATCHLST]
        [(OR (EQ (SETQ PROP (TABLE.ENTRY.GET
                                 (CAAR MATCHLST)))
                 (QUOTE ::))
             (AND (EQ (SETQ VAL
                           (TABLE.ENTRY.GET
                                 (CADAR MATCHLST)))
                      (QUOTE ::))
                  (MES.GETP MES PROP))
             (EQUAL (MES.GETP MES PROP) VAL)
             (AND (LISTP VAL)
                  (EQ (CAR VAL)(QUOTE ::))
                  (EVAL (CADR VAL))))


(* We finally get to the check itself. It will match in any
of the following cases:   1.0   the slot name is :: -- this
indicates that this part must always match...   2.0 or the
slot must exist in the message, and the value in the match
list is :: -- this means that we are only checking for the
existence of the slot.   3.0   the values of the slots
match in the message and the match list.   4.0 the value of
the slot is a list starting with :: then we just execute
the CDR of the list.
Then, if we were  in an OR list: we are through. Otherwise,
we must test the other elements of the upper list.)

          (OR ORFLG (MATCH.MES.WITH.TABLEENTRY MES
                                 (CDR MATCHLST]
        (ORFLG
```

```
        (* the match failed, but we were in an OR list: try
        the other elements)
                        (MATCH.MES.WITH.TABLEENTRY MES
                                                (CDR MATCHLST) T]
            (ORFLG
                        (* There is no more to match with, and we
                           were in an OR list: the match failed)
             NIL)
            (T
                        (* There is no more to match with, and we
                           were in an AND list: the match succeeded)
             T])


(MES.APPLY.ENTRY
  [LAMBDA (MES PROPVALLST)

        (* Creates the new message properties given in PROPVALLST)

        (* Each element of PROPVALLST is a list of 2 elements: a
           slot name, and a slot value. If the slot name is NIL,
           the slot value is still evaluated, but no new message
           slot is created -- supposedly in this case, we are only
           interested in the side effects of evaluating the slot
           value...)

    (PROG (PROP VAL (LOOPLST PROPVALLST)
                (NEWMES MES))
      LOOP(COND
            (LOOPLST (SETQ PROP (TABLE.ENTRY.GET (CAAR LOOPLST)))
                     (SETQ VAL (TABLE.ENTRY.GET (CADAR LOOPLST)))
                     (AND PROP (MES.PUT NEWMES PROP VAL))
                     (SETQ LOOPLST (CDR LOOPLST))
                     (GO LOOP))
            (T (RETURN NEWMES])



(TABLE.ENTRY.GET
  [LAMBDA (ELT)                    (* Evaluates the entry element)
    (COND
      ((OR (ATOM ELT)
           (NEQ (CAR ELT)
                (QUOTE :)))

        (* If the element is an atom: return it. If it is a list
        whose first element is not a : return it too. Otherwise,
        evaluate each element of the list)
```

```
  ELT)
(T (PROG ((ELTLST (CDR ELT))
          RESPONSE)
     LOOP(COND
           (ELTLST (SETQ RESPONSE (EVAL (CAR ELTLST)))
                   (SETQ ELTLST (CDR ELTLST))
                   (GO LOOP))
           (T (RETURN RESPONSE])
```

# Appendix D

## User Interface Functions

Appendix D

User Interface Functions

```
(TTYMES
 [LAMBDA NIL
   (PROG (MES ARGS PROP FN LOOPLST NUM)
         (SET.I.FLAG T)
         (TERPRI)
         (PRINT1 "Please specify the message properties and values:")
         (TERPRI)
         [SETQ MES (LIST (QUOTE TO)
                         (PROGN (TERPRI)
                                (SPACES 3)
                                (PRIN1 "TO")
                                (SPACES 13)
                                (PEEKC NIL T)
                                (CAR (READLINE)))
                         (QUOTE FROM)
                         (QUOTE TTY:)
                         (QUOTE MES.FUNCTION)
                         (QUOTE FN.MES)
        (* FN.MES is a function which knows how to call other
           functions It receives its arguments in a message where
           MES.FN is the name of the function to call; and it calls
           that function with the appropriate arguments.)
                         (QUOTE MES.FN)
                         (PROGN (SPACES 3)
                                (PRIN1 "MES.FN")
                                (SPACES 9)
                                (PEEKC NIL T)
                                (SETQ FN (CAR (READLINE]
         (SETQ ARGS (ARGLIST FN))
         (SETQ LOOPLST ARGS)
               (* ARGS is now the list of arguments of MES.FN)


    LOOP (SPACES 3)
         (COND
          (LOOPLST (SETQ PROP (CAR LOOPLST))
                   (PRIN1 PROP)
                  (* this prints the argument name)
                   (SPACES (COND
                             ((IGREATERP 14 (SETQ NUM (NCHARS PROP)))
                               (IDIFFERENCE 15 NUM))
                             (T 1)))
                   (PEEKC NIL T)
```

```
                (SETQ LOOPLST (CDR LOOPLST))
                (SETQ MES (APPEND MES (LIST PROP)))
                [SETQ MES (APPEND MES (COND
                                        ((READLINE))
          (* we get the argument value from the user)
                                        (T (LIST NIL]
                (GO LOOP))
           ((SETQ PROP (PROGN (PEEKC NIL T)
                              (READLINE)))
          (* we check if the user wants to add to the message)
             (SETQ MES (APPEND MES PROP))
             [OR (CDR PROP)
                 (SETQ MES (APPEND MES (LIST NIL]
             [SETQ ARGS (APPEND ARGS (LIST (CAR PROP]
                 (* if yes: we add it and its value to the message)
             (GO LOOP)))
         (TERPRI)
         (TERPRI)
         (PRINT1 "Thank you.")   (* it always pays to be polite)
         (SETQ MES (NCONC1 MES (QUOTE ARGS)))
         (SETQ MES (APPEND MES (LIST ARGS)))
         (TERPRI)
         (QUEUE.MES (APPLY (QUOTE CREATE.MES)
                           MES))
                (* finally the message is created and queued)
         (RETFROM (QUOTE USEREXEC])



(FN.MES
  [LAMBDA (MES)
    (PROG [LIST (ARGS (MES.GETP MES (QUOTE ARGS]
      LOOP(COND
            (ARGS [OR (EQ (CAR ARGS)
                          (QUOTE MES.FN))
                      (SETQ LIST (NCONC1 LIST (MES.GETP MES
                                                        (CAR ARGS]
                  (SETQ ARGS (CDR ARGS))
                  (GO LOOP)))
           (APPLY (MES.GETP MES (QUOTE MES.FN))
                  LIST]



(PRINT.TTYMES
  [LAMBDA (MES)
    (PROG (ID)
          (WRITE (MES.GETP MES (QUOTE MES.FUNCTION))
                 "message"
```

```
                (MES.GETP MES (QUOTE MES.ID))
                "received from"
                (MES.GETP MES (QUOTE FROM)))
          (TERPRI)
          (TERPRI)
      LOOP(SETQ ID (ASKUSER NIL NIL
                            "
Which action do you wish to take? "
                            (QUOTE ((R "eply to it
" RETURN (QUOTE R))
                                    (F "orward it
" RETURN (QUOTE F))
                                    (P "rint it
" RETURN (QUOTE P))
                                    (S "uspend
" RETURN (QUOTE S)
                                       EXPLAINSTRING
                                    "Suspend plan and save message")
                                    (D "efer answer
" RETURN (QUOTE D)
                                       EXPLAINSTRING
                  "Defer answer (message saved and plan continued)")
                                    (B "y-pass
" RETURN (QUOTE B)
                                       EXPLAINSTRING
               "By-pass question (plan continued but message lost)")
                                    (%
 "Reply to it
" NOECHOFLG T RETURN (QUOTE R)
 EXPLAINSTRING "<CR> may also be used to reply to the message.")))
                            T))
          (COND
            ((EQ ID (QUOTE P))
        (* the user wants to see the message: show it to him)
               (WRITE "It contains the following information:")
               (TERPRI)
               (for ELEMENT in MES do (WRITE (CAR ELEMENT)
                                             (CDR ELEMENT)))
               (GO LOOP))
             ((EQ ID (QUOTE R))
        (* the user wants to reply to the message: call REPLY.MES)
               (REPLY.MES MES))
             ((EQ ID (QUOTE S))
        (* the user wants to suspend planning: call SAVE.MES)
               (SAVE.MES MES))
             ((EQ ID (QUOTE B))
        (* the user wants to by-pass the question: call REPLY.MES
           to send back a null answer)
               (REPLY.MES MES T))
```

```
          ((EQ ID (QUOTE F))
        (* the user wants to forward the message: call FORWARD.MES)
           (FORWARD.MES MES))
          ((EQ ID (QUOTE D))
        (* the user wants to defer the answer: call SAVE.MES to save
           the message, then REPLY.MES to send back a null answer)
           (SAVE.MES MES)
           (REPLY.MES MES T])



(SAVE.MES                (* saves the message for later use.
                            asks for an ID under which it will
                            save the message, or generates one.)
  [LAMBDA (MES)
    (PROG (ID)
      LOOP[SETQ ID
            (ASKUSER NIL NIL "ID for saved message: "
                     (QUOTE (($ EXPLAINSTRING

"a number needed to save the message: this number may then be used
in REPLY.MES and FORWARD.MES later on.")
                            ($
 "" RETURN NIL EXPLAINSTRING
 "<CR> may also be used: then, the ID is automatically generated"
 NOECHOFLG T]
          [COND
            ((NULL ID)
              (PRINT1 (SETQ ID (SETQ MES.SAVED.ID (ADD1 MES.SAVED.ID]
          (COND
            ((NUMBERP ID)
              (PUTHASH ID MES MES.SAVED.HARRAY))
            (T (PRINT1 "?")
               (GO LOOP])



(REPLY.MES
  [LAMBDA (MES FLG)            (* builds a reply message to MES)
    (PROG (TTYFLG NEWMES ARGS)
          [COND
            ((NUMBERP MES)
              (SETQ TTYFLG T)   (* first get the saved message)
              (SETQ MES (GETHASH MES MES.SAVED.HARRAY]
          (COND
            (FLG (GO END)))
        (* if FLG is set, we just want to send a NULL message to
           continue the operations)
          [COND
            ((SETQ ARGS (MES.GETP MES (QUOTE ASK.TO.TTY)))
                (* a "specialized" function exists: use it)
```

```
            (SETQ NEWMES (APPLY* ARGS MES))
            (SETQ ARGS)
            (COND
              ([ASKUSER NIL NIL
               "
Do you want to add anything else to the message? "


                          (QUOTE ((Y "es
" RETURN T)
                                  (N "o
" RETURN NIL)
                                  (%
 "No
" NOECHOFLG T RETURN NIL EXPLAINSTRING
 "<CR> may also be used to say no."]
                (GO LOOP))
               (T (GO END]
          (COND
            ((SETQ ARGS (MES.GETP MES (QUOTE ARGS)))
        (* no specialized function: let us do it the hard way)
             (PRINT1 "Please, specify the following values
")))
      LOOP[COND
            (ARGS (SPACES 3)
        (* for each property in ARGS, ask a value to the user)
                (PRIN1 (CAR ARGS))
                (SPACES (COND
                          [(IGREATERP 15 (NCHARS (CAR ARGS)))
                            (IDIFFERENCE 15 (NCHARS (CAR ARGS]
                          (T 15)))
                (PEEKC NIL T)
              (* and put the property value pair in the message)
                (SETQ NEWMES (APPEND (LIST (CAR ARGS)
                                           (CAR (READLINE)))
                                     NEWMES))
                (SETQ ARGS (CDR ARGS))
                (GO LOOP))
            (T (SPACES 2)
               (* then ask for more property/value pairs)
               (PRIN1 "*")
               (PEEKC NIL T)
               (COND
                 ((SETQ ARGS (READLINE))
                  (SETQ NEWMES (APPEND ARGS NEWMES))
                  (SETQ ARGS)
                  (GO LOOP]
              (* we build the complete reply message)
      END (SETQ NEWMES (APPEND (LIST (QUOTE TO)
                                     (MES.GETP MES (QUOTE FROM))
                                     (QUOTE FROM)
                                     (OR (MES.GETV MES (QUOTE
```

```
                                                    ORIGINAL.FROM))
                                            (QUOTE TTY:))
                                    (QUOTE MES.FUNCTION)
                                    (MES.GETP MES
                                              (QUOTE RETURN.MES.TO))
                                    (QUOTE IN.RESPONSE.TO)
                                    MES)
                            NEWMES))
                (QUEUE.MES (APPLY (QUOTE CREATE.MES)
                                  NEWMES))
                     (* and queue the result)
                (COND
                  (TTYFLG (RETFROM (QUOTE USEREXEC])
```

We now give, as an example, the definition of REQUEST.ASSIGN.FROM.TTY which is the "specialized" function used in the ACS session of section V.

```
(REQUEST.ASSIGN.FROM.TTY
  [LAMBDA (MES)
    (PROG ((MODEL.NAME (MES.GETP MES (QUOTE MODEL.NAME)))
           (ID (MES.GETP MES (QUOTE REQUEST.ID)))
           (RES.NAME (MES.GETP MES (QUOTE RES.NAME)))
           (START (MES.GETP MES (QUOTE START)))
           (END (MES.GETP MES (QUOTE END)))
           (EST (MES.GETP MES (QUOTE EST)))
           (LST (MES.GETP MES (QUOTE LST)))
           (EET (MES.GETP MES (QUOTE EET)))
           (LET (MES.GETP MES (QUOTE LET)))
           X Y Z LST1)
          (TERPRI)
          (MAPPRINQ ("Need an assignment of a " RES.NAME " for "
                                        MODEL.NAME " number "
                                        ID "." TERPRI
                     "Current requirements are as follows:  "
                                        TERPRI))
          [COND
            ((LISTP START)
                        "Start time:  "
                        (CAR START)
                        "." TERPRI]
          [COND
            ((LISTP END)
              (MAPPRINQ ((TAB 5)
                        "End time:  "
                        (CAR END)
                        "." TERPRI]
          [COND
            (EST (MAPPRINQ ((TAB 5)
                        "Earliest start time desired:  "
                        (CAR EST)
```

```
                    "." TERPRI]
          [COND
            (LST (SETQ LST1 LST)
                 (MAPPRINQ ((TAB 5)
                            "Latest start time desired:  "
                            (CAR LST1)
                            "." TERPRI]
          [COND
            (EET (MAPPRINQ ((TAB 5)
                            "Earliest end time desired:  "
                            (CAR EET)
                            "." TERPRI]
          [COND
            (LET (MAPPRINQ ((TAB 5)
                            "Latest end time desired:  "
                            (CAR LET)
                            "." TERPRI]


          (TERPRI)
          (MAPPRINQ ((TAB 10)
                     "Enter name of " RES.NAME " assigned," TERPRI
                     (TAB 10)
                     "or <CR> to defer assignment:  "))
          [SETQ X
            (CAR (ASKUSER NIL NIL ""
                          (QUOTE ((%
NIL RETURN NIL EXPLAINSTRING "<CR> to defer assignment")
                                  ($ NIL RETURN ANSWER EXPLAINSTRING
                                       "name of a resource"]
          [COND
            ((NULL X)
              (RETURN NIL))
            [(AND (LISTP START)
                  (LISTP END))
              (COND
                ([ASKUSER NIL NIL "Are the start and end times OK? "
                          (QUOTE ((Y "es
" RETURN T)
                                  (N "o
" RETURN NIL)
                                  (%
 "yes
" RETURN T NOECHOFLG T EXPLAINSTRING "<CR> may also be used for yes"]
                 (SETQ Y (CAR START))
                 (SETQ Z (CAR END))
                 (GO L]
             ((AND (NULL (LISTP START))
                   (NULL (LISTP END))
                   LST EET)
              (COND
                ([ASKUSER NIL NIL
```

```
                "Are the latest end and earliest start times OK? "
                        (QUOTE ((Y "es
" RETURN T)
                                (N "o
" RETURN NIL)
                                (%
 "yes
" RETURN T NOECHOFLG T EXPLAINSTRING "<CR> may also be used for yes"]
              (SETQ Y (CAR LST))
              (SETQ Z (CAR EET))
              (GO L]
       (MAPPRINQ ((TAB 10)
                  "Enter start time of assignment:  "))
       (SETQ Y (READ))
       (MAPPRINQ ((TAB 10)
                  "Enter end time of assignment:  "))
       (SETQ Z (READ))
   L   (RETURN (LIST (QUOTE NAME)
                     X
                     (QUOTE START)
                     Y
                     (QUOTE END)
                     Z])



(FORWARD.MES                    (* forward the message)
  [LAMBDA (MES)
    (PROG (TTYFLG NEWMES)
          [COND
            ((NUMBERP MES)      (* first get the message itself)
              (SETQ TTYFLG T)
              (SETQ MES (GETHASH MES MES.SAVED.HARRAY]
          (PRIN1 "To: ")
          (PEEKC NIL T)         (* get the destination from the user)
          (MES.PUT MES (QUOTE TO)
                   (CAR (READLINE)))
          [COND
            ((ASKUSER NIL NIL
             "Do you want the originator of the message to be TTY:? "
                      (QUOTE ((Y "es
" RETURN T)
                              (N "O
" RETURN NIL)
                              (%
 "Yes
" NOECHOFLG T RETURN T EXPLAINSTRING
                      "<CR> may also be used for yes.")))
                      T)
             [MES.PUT MES (QUOTE ORIGINAL.FROM)
                      (OR (MES.GETP MES (QUOTE ORIGINAL.FROM))
                          (MES.GETP MES (QUOTE FROM]
```

```
    (MES.PUT MES (QUOTE FROM)
             (QUOTE TTY:]
      (* add an ORIGINAL.TO and a FROM part, then queue it)
(QUEUE.MES MES)
(COND
  (TTYFLG (RETFROM (QUOTE USEREXEC])
```

DISTRIBUTION LIST

| | |
|---|---|
| Defense Documentation Center<br>Cameron Station<br>Alexandria, Virginia 22314 | 12 copies |
| Office of Naval Research<br>Information Systems Program<br>Code 437<br>Arlington, Virginia 22217 | 2 copies |
| Office of Naval Research<br>Code 102IP<br>Arlington, Virginia 22217 | 6 copies |
| Office of Naval Research<br>Branch Office, Boston<br>495 Summer Street<br>Boston, Massachusetts 02210 | 1 copy |
| Office of Naval Research<br>Branch Office, Chicago<br>536 South Clark Street<br>Chicago, Illinois 60605 | 1 copy |
| Office of Naval Research<br>Branch Office, Pasadena<br>1030 East Green Street | 1 copy |
| New York Area Office<br>715 Broadway - 5th Floor<br>New York, New York 10003 | 1 copy |
| Naval Research Laboratory<br>Technical Information Division<br>Code 2627<br>Washington, D.C. 20375 | 6 copies |
| Dr. A. L. Slafkosky<br>Scientific Advisor<br>Commandant of the Marine Corps<br>Code RD-1<br>Washington, D.C. 20380 | 1 copy |
| Office of Naval Research<br>Code 455<br>Arlington, Virginia 22217 | 1 copy |
| Office of Naval Research<br>Code 458<br>Arlington, Virginia 22217 | 1 copy |

Naval Ocean Systems Center — 1 copy
Advanced Software Technology Division
Code 822
San Diego, California 92152

Mr. E. H. Gleissner — 1 copy
Naval Ship Research & Dev. Center
Computation and Mathematics Dept.
Bethesda, Maryland 20084

Captain Grace M. Hopper — 1 copy
NAICOM/MIS PLANNING BRANCH (OP-916D)
Office of Chief of Naval Operations
Washington, D.C. 20350

Mr. Kin B. Thompson — 1 copy
Technical Director
Information Systems Division (OP-91T)
Office of Chief of Naval Operations
Washington, D.C. 20350

Director — 1 copy
National Security Agency
Attn: Mr. Glick
Fort George G. Meade, Maryland 20755

Naval Aviation Integrated Logistic — 1 copy
Support Center
Code 800
Patuxent River, Maryland 20670

Professor Omar Wing — 1 copy
Columbia University in the
City of New York
Department of Electrical Engineering
and Computer Science
New York, New York 10027

Mr. M. Culpetter — 1 copy
Code 183
Naval Ship Research and
Development Center
Bethesda, Maryland 20084

Mr. D. Jefferson — 1 copy
Code 188
Naval Ship Research and
Development Center
Bethesda, Maryland 20084

Robert C. Kolb, Head — 1 copy
Code 824
Tactical Command Control and
Navigation Division
Naval Ocean Systems Center
San Diego, California 92152

Defense Mapping Agency Topographic Center — 1 copy
Attn: Advanced Technology Division
Code 41300 (Mr. W. Mullison)
6500 Brookes Lane
Washington, D.C. 20315

Commander, Naval Sea Systems Command — 1 copy
Department of the NAvy
Attn: PMS 30611
Washington, D.C. 20362

Professor Mike Athans — 1 copy
MIT
Dept. of Elec. Eng. & Comp. Science
77 Massachusetts Avenue
Cambridge, Massachusetts 02139

Captain Richard L. Martin — 1 copy
Cmd. Officer, USS Francis Marion
LPA-249
FPO, New York 09051